**Київський національний університет імені Тараса Шевченка**

**Факультет комп’ютерних наук та кібернетики**

**Кафедра інтелектуальних програмних систем**

**Алгоритми та складність**

**Завдання № 6**

**Варіант № 3**

**Звіт**

**Виконав:**

студент групи К-29

Грищенко Юрій Анатолійович

**Київ-2020**

**Завдання**

Реалізуйте B+ -дерево.

**Предметна область**: відділ кадрів (варіант 3). Об’єкти: відділення фірми, працівники. Маємо множину відділень, у кожному відділенні зберігається множина працівників.

**Теорія**

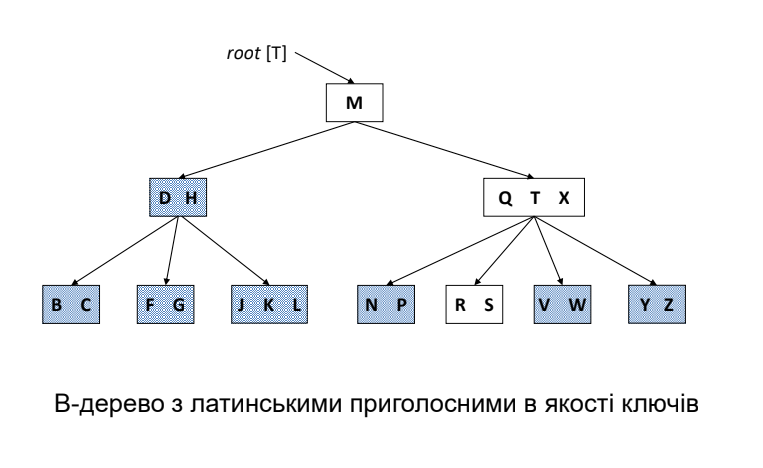
Припустимо, що у нам необхідно зберігати інформацію про велику кількість працівників відділення. Множина має бути відсортована в алфавітному порядку, і елементів надто багато, щоб одночасно зберігати їх усіх в оперативній пам’яті.

Для вирішення такої задачі варто скористатися **B+-деревом.** Взагалі, їх часто використовують для створення файлових систем, а також реляційних баз даних.[1]

Спершу розглянемо B-дерева, потім покажемо, чим відрізняються від них B+ -дерева.

**B-дерева:**

* Робота з великими об’ємами даних, які не поміщаються в оперативну пам’ять і зберігаються на диску (наприклад, СУБД, файлові системи).
* В-дерева – узагальнення бінарних дерев пошуку.
* Висока степінь розгалуження – вузли можуть мати до тисяч потомків.
* Якщо внутрішній вузол містить n[x] ключів, то він має (n[x]+1) синів.
* Ключі у вузлі x є роздільниками діапазону ключів на (n[x]+1) піддіапазонів.
* При пошуку переходимо до сина з потрібним діапазоном.



Час доступу до оперативної та вторинної пам’яті відрізняється в тисячі разів. Аналізуючи алгоритми, що враховують роботу над даними у зовнішній пам’яті, доцільно розглядати

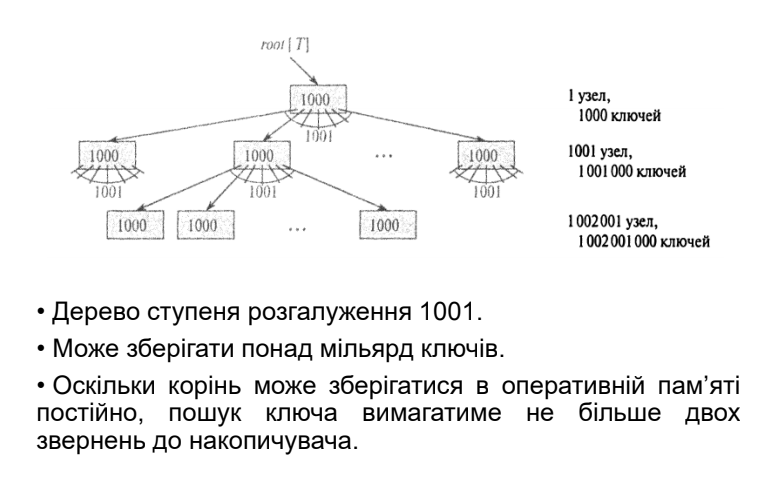
* кількість звернень до диску,
* час обчислень (процесорний час).

Інформація розбивається на сторінки однакового розміру (211-214 байт), що зберігаються послідовно в межах одного циліндру. Відповідно кожна операція читання або запису працює з декількома сторінками одночасно.

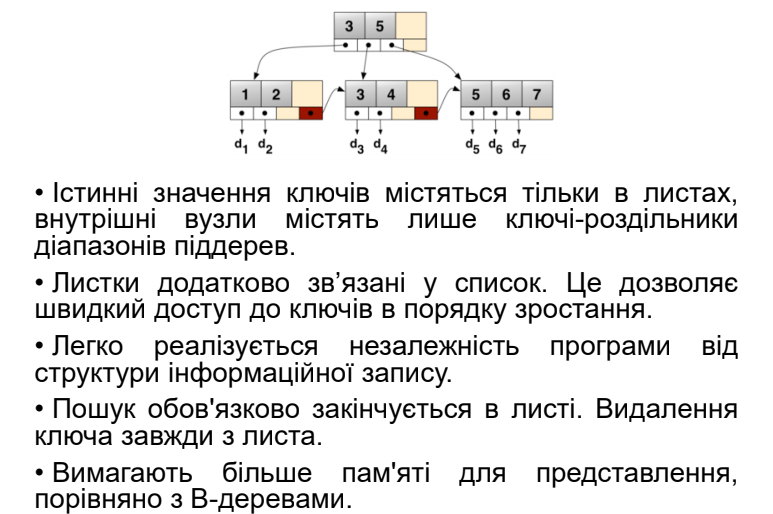
Алгоритми роботи над В-деревами копіюють до оперативної пам’яті тільки необхідні для роботи сторінки і записують назад лише змінені сторінки.

В довільний момент часу алгоритм працює над деякою постійною кількістю сторінок в оперативній пам’яті – а розмір самого В-дерева при цьому не обмежений.

Зазвичай для великих В-дерев ступінь розгалуження становить від 50 до 2000. Чим більше розгалуження, тим менша висота дерева, а отже і кількість звернень до диску.



**B+ -дерева —** модифікація В-дерева:

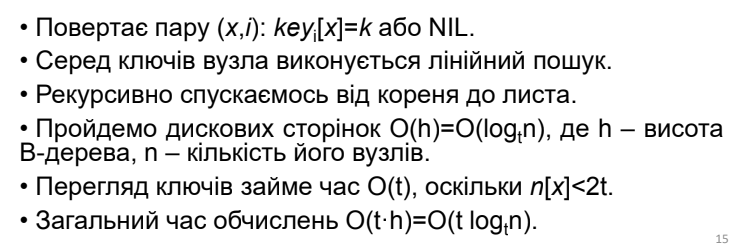


**Алгоритми**

Основні дії над B+ -деревом — це пошук, додавання та видалення елементів.

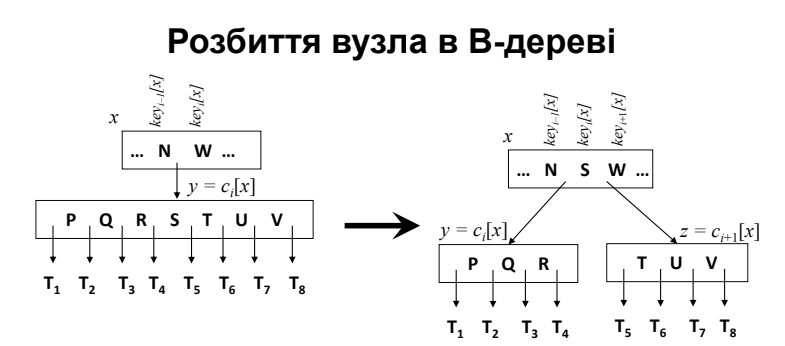
**Пошук:**

У B+ -дереві пошук виконується так само, як у звичайному n-арному дереві пошуку, але елемент шукається тільки серед листів.



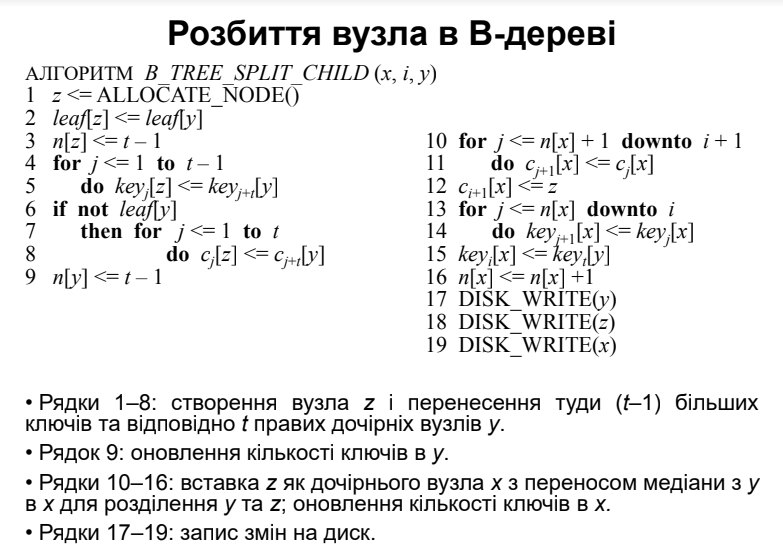
**Додавання елемента:**

* Замість вставки в потрібне місце листа, як у бінарному дереві пошуку, вставляємо новий ключ в існуючий лист.
* Якщо лист вже повний, він розбивається на два, а ключ, по якому відбувається розбиття, вставляється в батьківський вузол.
* Цей вузол також може виявитися повним, тому процес розбиття може “підніматися” до кореня.
* Вставку ключа можливо здійснити за один прохід від кореня до листа, якщо по ходу робити розбиття відвіданих заповнених вузлів, включаючи лист.



Вхідні параметри – незаповнений внутрішній вузол x, індекс i, та вузол y=ci[x] – заповнений син вузла x.

Дочірній вузол розбивається на два з (t–1) ключами кожний, а медіана вноситься у батьківський вузол разом з відповідною інформацією, включаючи інформацію про нового сина. Всі ключі y, більші за медіану, поміщаються в новий вузол z.



**Зауваження:***при роботі з В+ -деревами, на відміну від звичайного В-дерева, під час операції SPLIT ключі ніколи не “забираються” з листків. Тобто замість рядка 9 повинна бути перевірка*

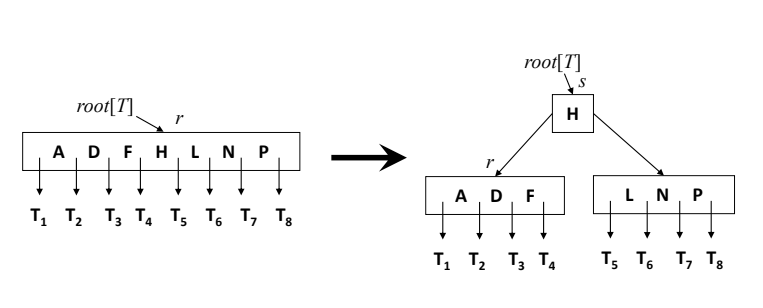
if leaf[y] then n[y] <= t else n[y] <= t-1

*Також, якщо y і z — листи, то в певний момент варто оновити значення next:*

if leaf[y]

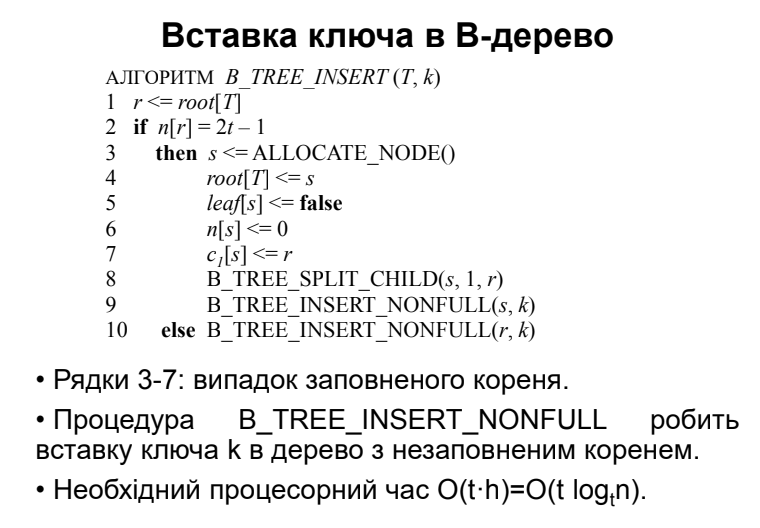
then next[z] <= next[y]; next[y] <= z

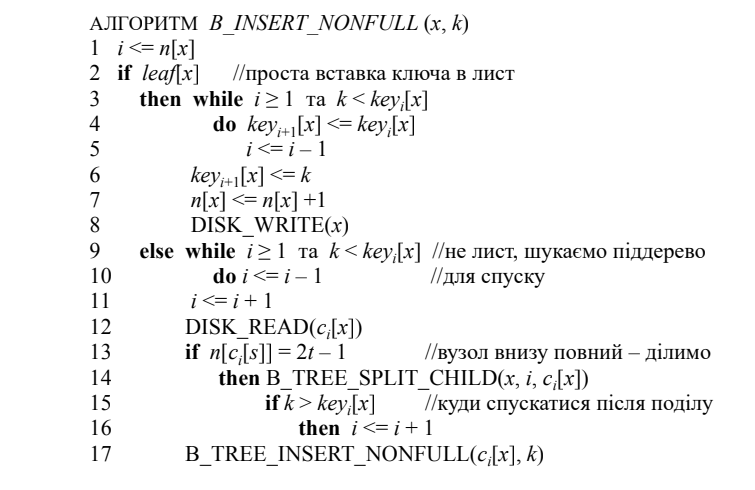
Виконується **O(1) дискових операцій**. Процедура перерозподіляє всі (2t–1) ключів, тому **час її роботи Θ(t)**.



Для розбиття заповненого кореня спочатку робимо його сином нового порожнього кореня. При цьому висота дерева збільшиться на одиницю.

Розбиття – єдиний спосіб збільшення висоти В-дерева. Дерево “підростає” зверху, а не знизу.



Процедура **B\_INSERT\_NONFULL** виконує або просту вставку в лист (що займає час О(t), або ж пошук піддерева, який теж займає час O(t). Тому **загальний час виконання — O(t).**

**Видалення:**

* Видалення ключа з В-дерева є складнішим за вставку.
* *Ключ видаляється тільки з листа.* Проте слід перевіряти, чи достатньо заповнені вузли (крім кореня).
* Щоб виконувати видалення ключа за один прохід, для роботи процедури потрібна сильніша умова: при виклику для вузла x там міститься не менше t ключів. Тому перед рекурсивним викликом можливе переміщення ключа в синівський вузол.
* В результаті видалення ключа висота дерева може зменшитися на 1.

**Процедура B\_PLUS\_DELETE(k,x). Випадки розгляду.**

1. Якщо ключ k знаходиться у вузлі x та останній є листом – видаляємо k з x.
2. Якщо ключ k відсутній у внутрішньому вузлі x, знаходимо корінь ci[x] піддерева, що має містити k (якщо такий ключ існує). Якщо ci[x] містить лише (t–1) ключ, виконуємо кроки 2а)-2г) для гарантії переходу у вузол з мінімум t ключами.

Потім рекурсивно видаляємо k з піддерева ci[x].

а) **якщо ci[x] є внутрішнім вузлом і містить тільки (t–1) ключ**, але один з його безпосередніх сусідів (брати справа і зліва, відділені єдиним ключем-роздільником) має хоча б t ключів, передамо ci[x] відповідний ключ-роздільник, на його місце поставимо крайній ключ сусіда і перенесемо відповідний вказівник від сусіда до ci[x].

б) **якщо ci[x] є листом і містить тільки (t–1) ключ**: передамо ci[x]крайній ключ сусіда і замінимо відповідний ключ-роздільник на менший серед крайніх ключів.

в) **якщо ci[x] є внутрішнім вузлом і якщо ci[x] з обома прямими сусідами містять по (t–1) ключу**, об’єднаємо ci[x] з одним з цих сусідів (при цьому відповідний ключ-роздільник стане медіаною нового вузла).

г) **якщо ci[x] є листом і якщо ci[x] з обома прямими сусідами містять по (t–1) ключу**, об’єднаємо ci[x] з одним з цих сусідів (при цьому відповідний ключ-роздільник видаляємо). Оновлюємо значення next.

B\_PLUS\_DELETE виконається в один прохід униз. Для В+ -дерева висоти h **потрібно О(h) дискових операцій**. Загальний процесорний **час** **O(t·h)=O(t logtn).**

**Основні модулі програми**

**Мова програмування: C++14.**

У своїй програмі я розбив модулі на дві категорії: ті, що стосуються саме предметної області (HumanResources.\*) і ті, що стосуються В+-дерева (BPlusTree.hpp). main.cpp містить код для інтерактивного режиму.

BPlusTree<T> реалізовано як template-клас, тобто він може зберігати обєкти будь-якого класу. Єдина умова: для класу Т мають бути реалізовані оператори порівняння (>. ≥, <. ≤), за допомогою яких будуть сортуватися елементи, і оператор <<, за допомогою якого елементи дерева виводяться на екран.

Зберігання та зчитування даних з зовнішньої пам’яті не реалізовано. Програма працює лише з оперативною пам’яттю. Також ця реалізація не підтримує елементів-дублікатів.

BPlusTree<T>::BPlusTree(size\_t t)

- конструктор дерева. в якому задається значення t.

BplusTree<T>::~BPlusTree()

- деструктор, звільняє пам’ять.

void BPlusTree<T>::insert(const T& element)

- вставляє елемент, виконує алгоритм B\_TREE\_INSERT.

void BPlusTree<T>::remove(const T& element)

- видаляє елемент, виконує алгоритм B\_PLUS\_DELETE.

bool BPlusTree<T>::contains(const T& element) const

- шукає елемент у дереві, повертає true якщо element міститься у листках дерева.

void BPlusTree<T>::printInOrder(std::ostream& os) const

- друкує всі елементи у дереві у відсортованому вигляді. Спочатку спускається до найменшого листка, і далі використовує вказівник next для обходу листів.

void BPlusTree<T>::printStructure(std::ostream& os) const

- друкує елементи дерева у стовбчик, з відступами, щоб візуально показати внутрішню структуру дерева.

**Інтерфейс користувача, тестові приклади**

У програмі є інтерактивний режим, який дозволяє користувачеві створити В+-дерево з будь-яким значенням t > 1, додавати, видаляти елементи і виводити на екран структуру дерева.

Enter value for t: **2**

Available commands: insert, delete, clear, print, find-by-name, help, exit

> **insert**

Enter employee name: **Yurii**

Inserted Yurii.

> **insert**

Enter employee name: **Bob**

Inserted Bob.

> **insert**

Enter employee name: **Alice**

Inserted Alice.

> **print**

[ { Name: Alice }, { Name: Bob }, { Name: Yurii } ]

== Tree ==

{ Name: Alice }

{ Name: Bob }

{ Name: Yurii }

> **insert**

Enter employee name: **Carl**

Inserted Carl.

> **print**

[ { Name: Alice }, { Name: Bob }, { Name: Carl }, { Name: Yurii } ]

== Tree ==

{ Name: Alice }

{ Name: Bob }

{ Name: Bob }

{ Name: Carl }

{ Name: Yurii }

> **insert**

Enter employee name: **Gary**

Inserted Gary.

> **insert**

Enter employee name: **Fred**

Inserted Fred.

> **insert**

Enter employee name: **Daniel**

Inserted Daniel.

> **insert**

Enter employee name: **Earl**

Inserted Earl.

> **print**

[ { Name: Alice }, { Name: Bob }, { Name: Carl }, { Name: Daniel }, { Name: Earl }, { Name: Fred }, { Name: Gary }, { Name: Yurii } ]

== Tree ==

{ Name: Alice }

{ Name: Bob }

{ Name: Bob }

{ Name: Carl }

{ Name: Daniel }

{ Name: Daniel }

{ Name: Earl }

{ Name: Fred }

{ Name: Fred }

{ Name: Gary }

{ Name: Gary }

{ Name: Yurii }

> **help**

Available commands: insert, delete, clear, print, find-by-name, help, exit

> **delete**

Enter employee name: **Carl**

Removed Carl.

> **print**

[ { Name: Alice }, { Name: Bob }, { Name: Daniel }, { Name: Earl }, { Name: Fred }, { Name: Gary }, { Name: Yurii } ]

== Tree ==

{ Name: Alice }

{ Name: Bob }

{ Name: Bob }

{ Name: Daniel }

{ Name: Daniel }

{ Name: Earl }

{ Name: Fred }

{ Name: Fred }

{ Name: Gary }

{ Name: Gary }

{ Name: Yurii }

> **delete**

Enter employee name: **Yurii**

Removed Yurii.

> **print**

[ { Name: Alice }, { Name: Bob }, { Name: Daniel }, { Name: Earl }, { Name: Fred }, { Name: Gary } ]

== Tree ==

{ Name: Alice }

{ Name: Bob }

{ Name: Bob }

{ Name: Daniel }

{ Name: Daniel }

{ Name: Earl }

{ Name: Fred }

{ Name: Fred }

{ Name: Gary }

> **find-by-name**

Enter employee name: **Daniel**

Tree contains Daniel? true

> **find-by-name**

Enter employee name: **Yurii**

Tree contains Yurii? false

> **exit**

**Висновки**

Ми дослідили структуру і властивості В-дерев та В+-дерев і зрозуміли, чому їх часто використовують для таких задач, як реляційні бази даних. Завдяки ним можна отримати оптимальну швидкість пошуку, вставки та видалення елементів з великих множин, мінімізуючи витрати часу на роботу з зовнішньою пам’яттю.

**Список використаних джерел**

1. https://en.wikipedia.org/wiki/B%2B\_tree
2. Лекція 4 з курсу “Алгоритми та складність” Шкільняк О.С. 2019-2020